# CSci 4203 Spring 2019 Lab Assignment 2 :

The purpose of this lab assignment is to learn how the MIPS pipeline works by modifying a given incomplete behavioral implementation of the MIPS pipeline from section 4.13 of the textbook (http://booksite.elsevier.com/9780124077263/downloads/advance_contents_and_appendices/section_4.13.pdf) and running test programs (benchmarks) on the pipeline.

The MIPS pipeline consists of five stages, namely, the IF (instruction fetch)), ID (instruction decode), EX (execute), MEM (memory) and WB (writeback). These stages have been implemented in verilog (partially) for a few benchmarks.

Pipeline Implementation :
All the verilog logic has been implemented in a single verilog file, cpu.v.
The pipeline components (pipeline registers, hold registers, control signals for forwarding etc.) are defined at the start of the program. Control signals are assigned for different pipeline stages, including those for bypassing and stalling. Next, the initial memory and register state is initialized inside an 'initial' block. Lastly, the pipeline stages are implemented inside an 'always @( posedge clock)' block, which causes the state to update every positive clock edge.

Benchmarks :
Eight benchmarks have been provided. Each benchmark is a 'unit test' for the pipeline and may test whether a single instruction, or a sequence of instructions, works correctly. The README file for the benchmarks briefly describes benchmarks. The same README file is also found at the end of this document for your convenience. Each benchmark contains 5 ".dat" files, namely, 'dmem.dat', 'imem.dat', 'regs.dat', 'mem_result.dat' and 'regs_result.dat'. The first three dat files represent the initial state of the memory and registers, before executing the benchmark. 'dmem.dat' contains the data, 'imem.dat' contains the instructions and 'regs.dat' contains the registers.

The .dat files contain a sequence of 32-bit fields, one field per line.
Each field represents a register value or a memory value, where the line number corresponds to the register number or the memory location. In total, there are 32 registers and 1024 memory location. Each register and memory location contains a 32-bit value. The values in imem.dat are binary encoded MIPS instructions.

In most examples, only the first few register and memory values are specified in the .dat files. For example, in benchmark 1, only the first 2 registers, only the first instruction and only the first 6 data memory values are specified. The remaining values are initialized to zero by cpu.v.
When a benchmark is executed, the initial state of the memory and registers is initialized using dmem.dat and regs.dat. The instructions in imem.dat are executed. The final state is printed in mem_result.dat and regs_result.dat.

Executing the Benchmarks on the Command Line :

Copy the four .dat files to the same folder as cpu.v. For example, copy these files from the Benchmarks/Benchmark1 directory into the same directory as cpu.v. Then execute the benchmark using the command './run.sh'. The result of execution will be printed in two newly created files, mem_result.dat and regs_result.dat.

Executing the Benchmarks using Vivado :

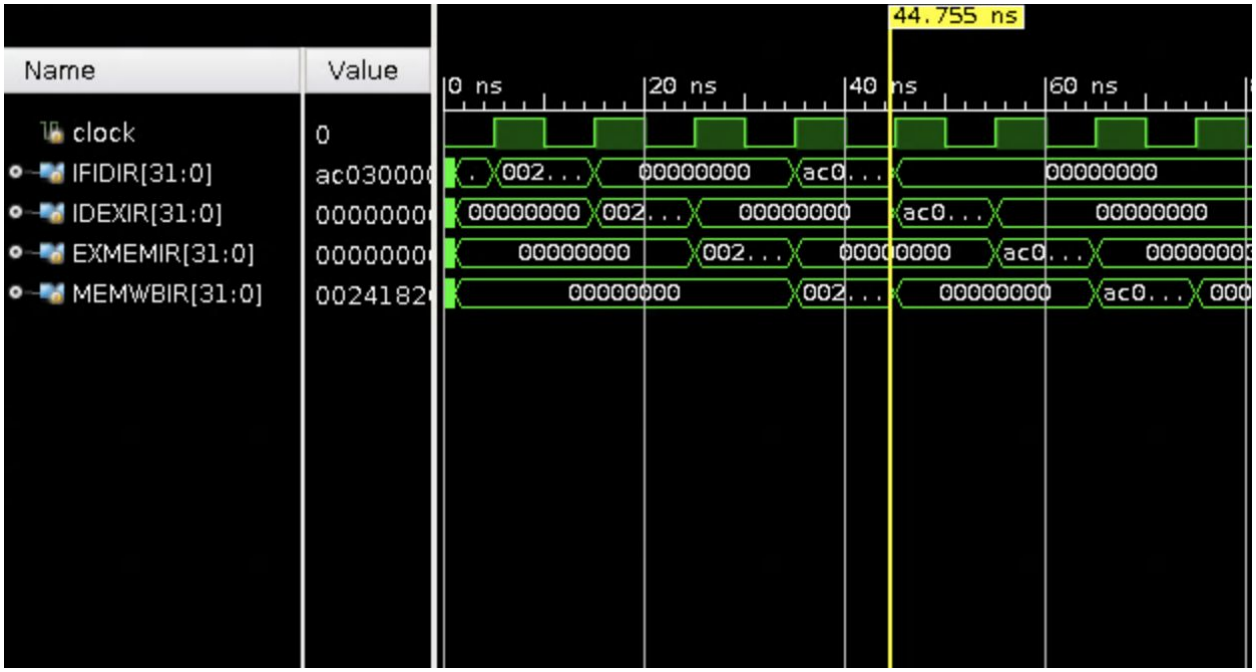
Import the cpu.v, tb_cpu.v to the project. Also import the four .dat files that you wish to use.
Change the path of these .dat files in cpu.v to the *absolute* path of these .dat files.
These paths can be changed in the "fopen" statements in cpu.v. Also, specify the *absolute* path of mem_result.dat and regs_result.dat in a similar fashion. Then execute the benchmark using 'Run Behavioral Simulation', similar to lab 1.The result of execution will be stored in two newly created files mem_result.dat and regs_result.dat, at the location specified by the path.

Understanding the execution :

The execution can be debugged by observing the propagation of data and instructions through the pipeline stages, and the corresponding control signals . Each instruction passes through all five pipeline stages, over the course of 5 or more clock cycles.

**Exercise** :

Execute one of the benchmarks using vivado, using 'Run Behavioral Simulation'.
In your waveform window, drag and drop the 'clock' control signal, the pipeline intermediate register signals, IFIDIR, IDEXIR, EXMEMIR and MEMWBIR, and the program counter PC, into the waveform window. Then, run 'Relaunch Simulation' to populate these waveform shapes.
Below is a screenshot of the result after doing so for a random benchmark with two instructions. The two instructions are propagating through the intermediate registers every positive clock cycle edge.




The four intermediate registers shown here are IFIDIR, IDEXIR, EXMEMIR, MEMWBIR. These intermediate pipeline registers store the instructions in binary form as they propagate through the pipeline. IFIDIR is between the IF (fetch) and ID (decode) stage, IDEXIR is between the ID and EX (execute) stages, EXMEMIR is between the EX and MEM (memory) stages, whereas MEMWBIR is between the MEM and WB (writeback) stages.
The instruction '002…' starts the IF stage at 5 ns and has propagated to the WB stage at 43ns, whereas the instruction 'ac0…' starts the IF stage at 25 ns has propagated to the WB stage in 74 ns. Add other pipeline registers, such as EXMEMALUOut (the result of the ALU), Ain, Bin (the inputs to the ALU), control signals such as bypassAfromMEM (forward from MEM to EX) etc. to the waverform to understand their functionality and verify their correctness.

**Check correctness of the execution** :
The result of the execution can be used to check correctness. For example, benchmark 1 contains a single 'lw' instruction. This instruction executes 'lw r5, 0(r1)'. r1 is initialized to 0x1, as specified by regs.dat. The memory location used by lw is 0x1 >> 2 = 0, since the memory locations are 4-byte aligned. The value in memory location 0 is 1, as specified in dmem.dat. This value is loaded into r5. regs_result.dat correctly shows that r5 has value 1.

***regs.dat*** has the following initialization:
r1 = 0x1

Memory location 0 in ***dmem.dat*** has the following initialization:
1

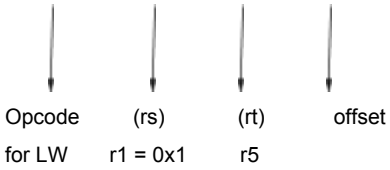
***imem.dat*** has the following initialization:

10001100001001010000000000000000

This represents an instruction word in binary for lw r5, 0(r1).

Let's break this into the fields of an instruction word in MIPS to understand it.

100011 00001 00101 00000 00000 000000




***regs_result.dat*** shows the final output of the register file - result after executing the instructions from the benchmark program.
r5 = 0x1

**Problems :**
The MIPS pipeline needs to be augmented so it can run other instructions.

Problem 0 (20 points):
Augment the given MIPS model so that it can run OR, AND, SLT and SLLV(Shift Left Logical Variable) instructions.
You need to modify the EX stage so that the ALU performs these operations.

You can refer to the following link for the Opcodes:
https://opencores.org/projects/plasma/opcodes

Also, an example of ADD instruction has been shown in the cpu.v file.

Problem 1 (40 points):
Augment the given MIPS model so that it can run the "conditional move" instruction(opcode 0, funct 29), which uses the 'R' instruction format and is defined as follows :

```
if (R[rs]<R[rt]) R[rd]=R[shamt]
 else do nothing
```

You need to modify the ID, EX, MEM and WB stages. The key modification is in the EX stage,, which compares the values of R[rs] and R[rt]. Also, the ALU now has 3 inputs instead of 2.
The WB stage needs to be modified so that R[shamt] value is written to R[rd], depending on the result of the EX stage. One way to do this is to propagate a 'flag' value from the EX stage to the MEM and WB stages, which indicates the result of the comparison. Additional control signals

may be required to handle data hazards between different stages. There may be other ways to implement this instruction, be creative !

Problem 2 (40 points):
Augment the given MIPS model so that it can run the “bpdcr” instruction(opcode 34) which uses the ‘I’ instruction format and is defined as follows :

```
bpdcr $rs, loop

if (R[rs]>0) R[rs]=R[rs]-1 and PC=PC+4+loop
else do nothing
```

‘Loop’ refers to the immediate field, aligned to 4-byte value by right shifting by 2 ( i.e. >> 2).
You need to modify the ID, EX, MEM and WB stages.The branch result is resolved in the EX stage. Stall the pipeline until the branch is resolved. The PC value needs to be propagated to the EX stage so that the new PC value can be calculated during the EX stage.Once the branch is resolved, update the PC value and unstall the pipeline. The WB stage needs to update the value of R[rs] depending on the result of the EX stage. Therefore, you may need to propagate a ‘flag’ value from the EX stage to the MEM and WB stage, which indicates the result of the EX stage. Additional control signals may be required to handle data and control hazards. There may be other ways to implement this instruction, be creative !

**Benchmarks for Problem 0, Problem 1 and Problem 2 :**

A. The key benchmark should contain a single new instruction in order to test whether it can work correctly by itself, in the absence of data and control hazards, since there are no other instructions in the pipeline.
B. Other benchmarks should test whether the new instruction can work correctly in the presence of data and control hazards due to other instructions in the pipeline, such as lw, sw and add.
C. Lastly, some credit will be assigned to ensure that your implementation does not break the provided benchmarks.

Handout :
You are provided with an incomplete MIPS behavioral model, a testbench, example benchmarks and this pdf. The benchmark folder has a “README_benchmarks.md” text file which briefly describes each benchmark. You are also provided with a script “run.sh” which compiles and runs the simulation on the command line.

Handin :
Place your new cpu.v file and any other verilog files (except the testbench) in a new folder, called Lab2. Compress Lab2 into a tar.gz file, which should have the name Lab2_<name>.tar.gz.

This can be done using the terminal command

tar -czvf Lab2_<name>.tar.gz Lab2_<name>

For example, I would turn in the file ‘Lab2_Kartik.tar.gz’.

Grading Criteria :
I will test your code using the provided benchmarks as well as others not available to you. Your final grade will depend on the fraction of benchmarks which execute correctly.

**Important** :
Please ensure that your code can be run on the command line using ‘./run.sh’.
Due to the large number of students, the grading will be done automatically and no credit will be assigned if your code does not compile.

**Benchmark Description :**

Benchmark 1 :
The first benchmark tests 'lw' instruction.
lw r5, 0(r1)

Benchmark 2 :
The second benchmark tests 'sw' instruction.
sw r3, 0(r0)

Benchmark 3 :
The third benchmark tests 'add' instruction.
add $r1, $r4, r5

Benchmark 4 :
The fourth benchmark tests a sequence of add instructions with a
RAW (read-after-write) hazard and load, which is solved using forwarding
and a stall.

lw $r3, 0($r0)
lw $r4, 1($r0)
add $r5, $r3, $r4
add $r6, $r5, $r4

Benchmark 5 :
Tests forwarding between WB stage and ID stage for ALU operation.
add $r1, $r2, $r3

```
nop
nop
add $r5,$r1, $r2
```

Benchmark 6 :
Tests the BNE instruction by incrementing a register until it reaches a particular value.

```
add $r2, $r1, $r2
nop
nop
nop
bne $r2, $r6, -5
```

Benchmark 7 :
Tests hazard on SW rt register.

```
add $r3, $r1, $r4
sw $r3, 0($r0)
```

Benchmark 8 :
Tests hazard on SW rt register caused by a LW to that register.

```
lw $r3, 0($r0)
sw $r3, 4($r0)
```

Benchmark 9:
Tests the ‘SLLV’ operation

```
sllv $r3, r1, r2
```

This has to perform $r3 = $r1 << $r2